CEWES MSRC/PET TR/98-03

A Fortran90 Module for Message-Passing Applications with Unstructured Communication Patterns

by

S. W. Bova

01h01298

Work funded by the DoD High Performance Computing Modernization Program CEWES Major Shared Resource Center through

Programming Environment and Training (PET)

Supported by Contract Number: DAHC 94-96-C0002
Nichols Research Corporation

A Fortran90 Module for Message-Passing Applications with Unstructured Communication Patterns

S.W. Bova*

February 5, 1998

1 Introduction

For high-performance, scientific computing, message-passing is often the paradigm of choice, *e.g.* MPI, the Message-Passing Interface standard. Incorporating a message-passing capability within an application involves the consideration of several bookkeeping issues such as the number and identity of processors with which a given processor communicates, the identification of data which must be exchanged, *etc.*

Consider a finite element (or finite volume) application that solves a partial differential equation. In order to develop a message-passing parallel implementation, the finite element mesh must be partitioned among the processors. Furthermore, point-to-point communication is required as a result of the local nature of the finite element approximation. More specifically, each process must exchange data associated with grid points along the inter-processor interfaces.

If a structured mesh is used to discretize the domain, then the resulting point-to-point communication patterns are also structured. For example, a structured grid can easily be partitioned among the processors in such a way that, except possibly for processors along the boundaries, each has an upper, lower, left and right neighboring process. Contrast this situation to that which is encountered if an unstructured mesh is used to discretize the domain. Figure 1 illustrates an unstructured triangulation that has been partitioned among several processors. In general, each processor has a different number of processes with which it must exchange data, and the length of each message can also vary substantially. In order to manage such unstructured communication patterns, certain data must be available to the application. In particular, each processor must store the number of neighboring processors for point-to-point communication; the destinations (origins) of the messages it sends (receives); the number of grid points along each inter-processor boundary; the identities of these grid points; arrays (buffers) in which the incoming and outgoing messages are stored; and finally, the identity of the processor in question.

Many computational mechanics applications have unstructured communication patterns so that simply representing the data associated with a message is a cumbersome task. If the application is written in C or C++, the use of structures or classes provides a natural

*CEWES MSRC On-site CFD Lead for PET, Mississippi State University

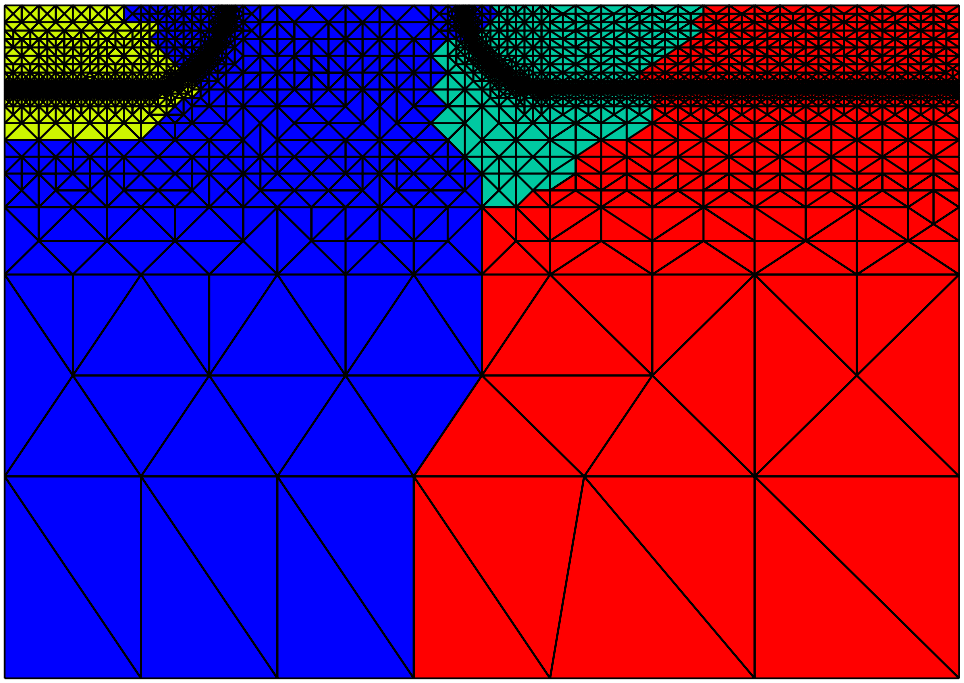

Figure 1: An unstructured finite element mesh. The colors represent data distributed among different processors.

way to organize this data. However, the majority of scientific applications are written in Fortran. The remainder of this work describes one way in which this data can be organized within a Fortran program which uses a finite element method to solve a partial differential equation. This approach exploits some of the new features of Fortran90. Since Fortran77 is a subset of Fortran90, existing Fortran77 codes can easily use the proposed approach by switching to a Fortran90 compiler and incrementally adding the new language features. This report assumes that the reader has a basic familiarity with message-passing computing and Fortran90. It is not intended to be a tutorial in either subject. The interested reader who would like more background information should consult the literature (*e.g.* see [1, 2]).

I emphasize that the type of unstructured communication patterns illustrated above may arise in applications other than unstructured finite element solvers. However, very similar data organization issues will need to be addressed. Hence the approach described herein may be applied. Features of Fortran90 such as modules, dynamic memory allocation, global variables, and user-defined datatypes are exploited in an attempt to bundle this data with functions in the spirit of a C++ class. (In fact, many object-oriented concepts can be expressed in Fortran90 [3, 4, 5]. See also the object-oriented Fortran90 web page at http://www.cs.rpi.edu/~szymansk/oof90.html.) It should be noted that the current MPI standard does not specify a Fortran90 binding. I describe below one way in which these language enhancements may be used in a portable MPI implementation which adheres to the standard. Section 2 describes the proposed data structure and associated subroutines; Section 3 illustrates how the module may be used; Finally, Section 4 summarizes the proposed approach and describes how to obtain the module.

2 Module Definition

2.1 Data Declarations

The module is written assuming that message-passing is implemented using an MPI library, but the module does not depend on MPI functionality nor does it reference any MPI files or routines. Thus, it should be straightforward to modify the module to use a different message-passing library. The key to the data organization is that the messages are stored in an array of user-defined datatypes. In the C language this would be an array of structures. This array, together with other variables such as the process rank, MPI communicator, *etc.*, are declared as global data (across the program units which are local to a single processor). This is in contrast to a Fortran77-style common block, in which the memory locations are made available to all the subprograms. With a common block, the same location in memory could in fact be referenced by many different variables. A Fortran90 module differs in that it introduces a global scope of a variable very similar to that of C programs.

The precise variable declarations are given in Figure 2. The `implicit none` statement turns off implicit typing, and forces explicit decalration of all variables. This statement helps to detect typographical errors in the source code at compilation instead of at run time. The next group of statements is the heart of the data organization strategy. This group defines a datatype called `r_message`. Here the number of grid points that make up the message, its destination/origin, and allocatable space for the send buffer, receive buffer, and the list of grid points are all bundled together. Next, an allocatable array of type `r_message` is

```
module mp_kit
  implicit none
  type r_message
     integer npoints            ! number of grid points
     integer dest               ! destination/origin of message
     real,  pointer, dimension(:) :: rbuff !receive buffer
     real,  pointer, dimension(:) :: sbuff !send buffer
     integer, pointer, dimension(:) :: points !list of grid points on
!             processor interface for a given message
  end type r_message

  type (r_message), allocatable, dimension(:) :: mesg

  integer :: mp_comm     ! mpi communicator
  integer :: mp_nsr      !number of neighboring processors for
!                         pt-to-pt communication
  integer :: mp_myid     ! rank of this process

  integer, allocatable, dimension(:,:):: mp_stats ! MPI message statuses.
  integer, allocatable, dimension(:) :: mp_reqst ! MPI request handles
end module mp_kit
```

Figure 2: Module data definitions of user-defined datatype and global variables.

declared and named `mesg`, followed by integer variables which are used to store the MPI communicator, the number of neighboring processes, and the rank of the process in question. Finally, two more allocatable arrays, `mp_stats` and `mp_reqst`, are declared. These arrays are intended to be used for the MPI status and MPI request arrays that are required when using the non-blocking communication routines `MPI_Isend()` and `MPI_Irecv()`. In MPI, each pending asynchronous send or receive operation is assigned a "request handle". An array is needed to simultaneously track more than one operation. Similarly, a completed operation has several pieces of information associated with it which are collectively termed a "status". (See [1] for the details on MPI asynchonous communication routines.)

2.2 The Subroutines

In order to provide functions and subroutines to operate on the data shown in Figure 2, the module is modified to include four subroutines as shown in Figure 3. The last argument of each routine is an error code that is returned from the Fortran90 `allocate` and `deallocate` statements. Thus the programmer can test the error code, and if nonzero, appropriate action can be taken. This work implements only the basic functions of message creation and deletion. Additional functionality could of course be provided; *e.g.* to load and unload message buffers. Below, I describe the basic purpose of each subroutine. Examples of their use are given in the following section.

```
module mp_kit
  implicit none

  ...previously defined variables...

contains
  subroutine mp_kit_init(ssize, nproc, comm, rank, error_code)
    integer, intent(in) :: ssize      ! mpi status size
    integer, intent(in) ::  nproc     !number of neighboring processors
    integer, intent(in) :: comm !mpi communicator
    integer, intent(in) :: rank ! rank of this process
    integer, intent(out) ::  error_code   !allocation error code from system
    ...rest of routine...

  subroutine alloc_msg(dumb, npoints, dest, dof, list, error_code)
    type (r_message) :: dumb    ! dummy message structure
    integer, intent (in) :: npoints     !number of grid points on the interface
    integer, intent (in) :: dest        !destination
    integer, intent (in) :: dof         !degrees of freedom per grid point
    integer, dimension(npoints), intent (in) :: list ! list of points
!                                        on processor interface
    integer, intent(out) ::  error_code   !allocation error code from system
    ...rest of routine...

  subroutine dealloc_msg(dumb, error_code)
    type (r_message), intent(in) :: dumb  ! dummy message structure
    integer, intent(out) ::  error_code   !deallocation error code from system
    ...rest of routine...

  subroutine mp_kit_close(error_code)
    integer, intent(out) ::  error_code   !deallocation error code from system
    ...rest of routine...

end module mp_kit
```

Figure 3: Module subroutine bindings.

The first subroutine, `mp_kit_init()`, initializes the module. All arguments except the last one are input. It sets the variables `mp_nsr`, `mp_comm`, and `mp_myid`. It also allocates the arrays `mp_stats`, `mp_reqst`, and `mesg`. The next routine, `alloc_msg()`, allocates the arrays `rbuff`, `sbuff`, and `points` for a single element of type `r_message`. It also intializes the fields `npoints`, `dest`, and `points`. The subroutine `dealloc_msg()` frees the memory associated with the arrays `rbuff`, `sbuff`, and `points`. Finally, the subroutine `mp_kit_close()` frees the memory associated with the arrays `mesg`, `mp_stats`, and `mp_reqst`.

3 Using the Module

The module is initialized by calling `mp_kit_init()` once. Values for the dummy arguments `ssize`, `rank` and `comm` are obtained from the message-passing system. Then each message structure in the array `mesg` is allocated and initialized by calling `alloc_msg()`. The routines `dealloc_msg()` and `mp_kit_close()` are also provided. (It may not be necessary to call these routines, since their only function is to free the memory associated with the message structures and this will happen automatically at program termination.) Figure 4 presents an example of how to initialize the module. Note that each program unit that accesses the module data or functions must issue the statement `use mp_kit`.

Finally, Figure 5 presents a code fragment which illustrates how a subroutine may load the message buffers and perform an asynchronous exchange. For each message, the number of words is calculated from the number of grid points on the interprocessor interface and the number of degrees of freedom per grid point. First, a non-blocking receive is posted. The rank, or process number, of the message destination is then extracted from the `r_message` datatype. (For an exchange, each processor must both send a message to and receive a message from each of its neighbors, thus the destination and origin of each send/receive pair is identical.) Next, a loop is performed over the interprocessor interface and each grid point index is extracted from the structure. The send buffer for the message is subsequently loaded from the solution array, and a call to `MPI_Isend()` is made to initiate the send. Finally, a call to `MPI_Waitall()` completes the exchange.

Some discussion is warranted regarding the precise syntax of the calls to `MPI_Irecv()` and `MPI_Isend()`. As mentioned previously, there is currently no Fortran90 binding for MPI. Thus, the MPI routines are simply expecting an address and have no capability to interpret user-defined structures as defined by Fortran90. Consider the syntax of `MPI_Isend()`. The first argument is the message buffer. In Figure 5 this buffer is denoted `mesg(i)%sbuff(1)`. In fact, since there are no restrictions on passing pointers to arrays as subroutine actual arguments, all of the following are equivalent[6]:

```
call MPI_Isend(mesg(i)%sbuff(1), words, MPI_REAL,  rank, &
     tag, mp_comm, mp_reqst(r_num), mpierr)

call MPI_Isend(mesg(i)%sbuff, words, MPI_REAL,  rank, &
     tag, mp_comm, mp_reqst(r_num), mpierr)

call MPI_Isend(mesg(i)%sbuff(:), words, MPI_REAL,  rank, &
     tag, mp_comm, mp_reqst(r_num), mpierr)
```

```
program myprog

  use mp_kit
...rest of routine...

! initialize MPI
  call MPI_Init(mpierr)

  call MPI_Comm_rank(MPI_COMM_WORLD,myid,mpierr)
  call MPI_Comm_size(MPI_COMM_WORLD,commsize,mpierr)

! initialize the module

  call mp_kit_init(MPI_STATUS_SIZE,nsr,MPI_COMM_WORLD,myid,ierr)

  ndof = 4 ! four degrees of freedom per grid point

! initialize the messages
  do i=1, nsr
     read(*,*)dest,nint !get destination and number of interface points
     do j=1, nint
        read(*,*) iface(j) !get the list of grid points on the interface
     end do
     call alloc_mesg(mesg(i), nint, dest, ndof, iface)

  end do
 ...rest of routine...
```

Figure 4: Illustration of how to initalize the module data.

```
  ...rest of routine...
  do i = 1, mp_nsr ! loop over the messages
     words =  ( mesg(i)%npoints )*ndof
     rank  = mesg(i)%dest

     call MPI_Irecv(mesg(i)%rbuff(1), words, MPI_REAL, &
          rank, tag, mp_comm, mp_reqst(i), mpierr)

!    load up the send buffer and send message
     k = 1
     do j = 1, mesg(i)%npoints
        jpt = mesg(i)%points(j) ! get the grid point number
        mesg(i)%sbuff(k) = soln(1,jpt)     !load first DOF
        mesg(i)%sbuff(k+1) = soln(2,jpt)   !load second DOF
        mesg(i)%sbuff(k+2) = soln(3,jpt)   !load third DOF
        mesg(i)%sbuff(k+3) = soln(4,jpt)   !load fourth DOF
        k = ndof*j + 1
     end do
     r_num = mp_nsr + i ! increment the request number

     call MPI_Isend(mesg(i)%sbuff(1), words, MPI_REAL,  rank, &
          tag, mp_comm, mp_reqst(r_num), mpierr)
  end do

  call MPI_Waitall(2*mp_nsr,mp_reqst, mp_stats, mpierr)

  ...rest of code...
```

Figure 5: Illustration of how an asynchronous, non-blocking exchange may be performed.

```
call MPI_Isend(mesg(i)%sbuff(1:words), words, MPI_REAL,  rank, &
     tag, mp_comm, mp_reqst(r_num), mpierr)
```

In practice, however, (at the time of this writing) there is a bug in version 7.1 of the SGI Fortran90 compiler. The result of this bug is that correct results are obtained only if the buffer is passed as `mesg(i)%sbuff(1)`. Using version 4.1.0.3 of the IBM compiler, all of the above cases work as expected.

4 Summary

A Fortran90 module has been presented which exploits new features of the language such as modules, dynamic memory allocation, global variables, and user-defined datatypes. Details of the data structure and associated subroutines were also described. This module has been used to parallelize a serial, unstructured finite element program [7]. Interested readers may obtain the module, together with associated makefiles (for the IBM SP, CRAY T3E, and SGI O2000) and a test program from `http://www.erc.msstate.edu/~swb/Tools`.

A primary advantage of using the approach of user-defined datatypes and modules is that the data required for message-passing are stored as global variables. Typically, the message-passing occurs inside several levels of subroutine calls. The use of modules allows this data to be available without modifying the argument list of all parent routines and simultaneously avoids introducing `common` blocks. Furthermore, the use of the user-defined datatype allows data to be organized in such a way that the resulting code is, in my opinion, more readable.

Future developments of the module are planned. For example, the `private` attribute could be added to the definition of the `r_message` datatype with functions to access certain elements by the user. In this way, data elements such as the message destination could be protected from corruption. This would be safer at the cost of a little overhead. Also, other modules could be defined which are based on the present one which would contain routines to pack/unpack message buffers, send messges, *etc.* This would make the resulting codes easier to read and maintain.

References

[1] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, London, 1994.

[2] Walter S. Brainerd, Charles H. Goldberg, and Jeanne C. Adams. *Programmers Guide to Fortran90*. Springer, New York, 1996.

[3] Viktor K. Decyk, Charles D. Norton, and Boleslaw K. Szymanski. Expressing object-oriented concepts in Fortran90. *ACM Fortran Forum*, 16(1), April 1997.

[4] Viktor K. Decyk, Charles D. Norton, and Boleslaw K. Szymanski. How to express C++ concepts in Fortran90. Technical Report PPG-1569, Institute of Plasma and Fusion Research, UCLA Dept. of Physics and Astronomy, Los Angeles, CA 90095-1547, February 1997. Submitted for publication.

[5] Viktor K. Decyk, Charles D. Norton, and Boleslaw K. Szymanski. Introduction to object-oriented concepts using Fortran90. Technical Report PPG-1560, Institute of Plasma and Fusion Research, UCLA Dept. of Physics and Astronomy, Los Angeles, CA 90095-1547, July 1996.

[6] Charles Koelbel. Rice University. Private communication, July 1997.

[7] S. W. Bova. A scalable SUPG method for free surface flows. Poster presentation at SC97, San Jose, CA, 1997.